A Compiler and Runtime Infrastructure for Automatic Program Distribution[*]

Roxana E. Diaconescu
California Institute of Technology
Center for Advanced Computing Research
roxana@caltech.edu

Lei Wang, Zachary Mouri, Matt Chu
University of California, Irvine
Department of Computer Science
{leiw, chum, zmouri}@uci.edu

Abstract

This paper presents the design and the implementation of a compiler and runtime infrastructure for automatic program distribution. We are building a research infrastructure that enables experimentation with various program partitioning and mapping strategies and the study of automatic distribution' s effect on resource consumption (e.g., CPU, memory, communication). Since many optimization techniques are faced with conflicting optimization targets (e.g., memory and communication), we believe that it is important to be able to study their interaction.

We present a set of techniques that enable flexible resource modeling and program distribution. These are: dependence analysis, weighted graph partitioning, code and communication generation, and profiling. We have developed these ideas in the context of the Java language. We present in detail the design and implementation of each of the techniques as part of our compiler and runtime infrastructure. Then, we evaluate our design and present preliminary experimental data for each component, as well as for the entire system.

1. Introduction

There are important potential benefits of automatic over manual program distribution, such as correctness, increased productivity, adaptive execution, concurrency exploitation. This paper describes a new approach to automatic program distribution. In contrast with previous work, instead of considering a particular class of programs and optimization targets, we consider general-purpose programs and study multiple optimization targets. Our system accepts a monolithic program and transforms it into multiple communicating parts in networked systems.

1.1. Possible Uses

Our approach places high emphasis on the generality of the distribution strategy and the ability to build an abstract model of the execution environment. Then, the distribution strategy can be specialized to concrete environments. We recognize that this approach may not be suitable for all computations. Many programs may not need distribution at all.

In some cases, however, automatic distribution is crucial. New technologies such as pervasive computing require that applications connect from any device, over any network, using any style of interface. Mobile computing requires that mobile code is deployed over heterogeneous networks of sometimes resource constrained devices. If there are not enough resources available to accommodate a given program on a single computing node, the promises of these technologies cannot be delivered. In this context, automatic distribution can help with increased accessibility, resource sharing, and load balancing.

Another broad class of data intensive applications relies on networked systems to process their data concurrently. Such applications range from inherently concurrent applications like image processing, universe exploration, computer supported cooperative work, to *loosely concurrent* applications such as fluid mechanics in avionics and marine structures. In this context, automatic distribution can help with exploiting concurrency, reducing the execution time, and increasing scalability.

Our specific technical contributions relative to previous systems with similar goals are:

- A set of techniques for a novel approach to automatic program distribution. These techniques are: object dependence graph construction, general graph partitioning, automatic communication generation, and automatically distributed program execution.

- An original compiler and runtime infrastructure that implements all the above techniques to allow flexible program distribution based on program access pattern, resource requirements, and resource availability.
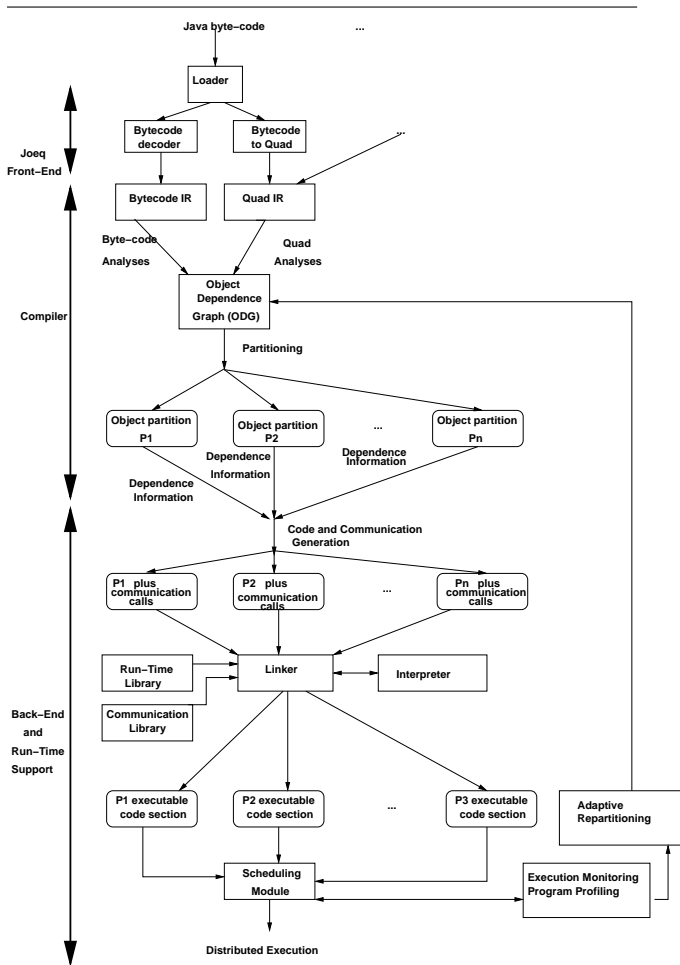
Figure 1. The distributed compiler and runtime infrastructure.

1.2. Basic Approach

Our compiler and runtime infrastructure is depicted in Figure 1. This system transforms sequential Java programs into distributed programs. Moreover, the system attempts to model the resources needed by the sequential program and distribute the program based on the resource availability in a networked system.To this effect, the system performs the following transformations:

1. The front-end transforms Java bytecode into the intermediate representation using Joeq front-end [22]. Joeq provides us with two intermediate representations: bytecode and *quad*. The latter is a quadruple style IR which resembles register-based representations.

2. The system uses our static analysis framework to ap-

proximate the object dependence graph for a program and model its resource requirements.

3. The system partitions the object dependence graph using a Java wrapper of the Metis graph partitioning tool [14].

4. The system uses bytecode rewriting to insert communication calls for remote dependences in the partitioned program. Also, the system uses a bottom-up rewrite system to generate target code for the various platforms making-up the networked configuration. For better resource utilization, in the future we plan to use native execution rather than Java Virtual Machine (JVM) hosted execution on (possibly resource-constrained) devices.

5. The system monitors the program execution and collects a set of statistics about resource usage. We use this information to gain insight into static partitioning. In the future we plan to use this information to perform adaptive repartitioning.

The rest of the paper is organized as follows. Section 2 describes the object dependence graph construction. Section 3 explains how we use graph partitioning to model resources and split the program into multiple pieces. Section 4 describes in detail our approach to code and communication generation. Section 5 presents the implementation of a runtime system that allows automatic distributed execution. Section 6 presents the design and implementation of a mixed instrumentation and sampling profiler that monitors programs during execution. Section 7 discusses an initial evaluation of the techniques we introduce in this paper. Section 8 reviews related research and contrasts our effort from previous approaches similar in goals. Section 9 concludes the paper and underlines future research directions.

2. Dependence Graph Construction

The first transformation our system performs is to create the dependence graph of the program. This graph depicts the dependences between program objects and serves as the input for the resource modeling and graph partitioning phase. We use a concrete example to illustrate the dependence graph construction.

2.1. An Example

Figure 2 shows an example of a Java program that we use throughout the paper. In our example, there are two classes. The `Account` class describes a bank account with a unique identifier, holder name, checking, savings, and loan. The `Bank` class describes a banking institution with a unique identifier, name, number of customers, and a list (`java.lang.Vector`) of their actual accounts.

```
public class Account {
}

public class Bank {
  protected Bank(String name, int numCustomers, int initialBalance,) {
    ...;
    initializeAccounts(initialBalance);
  }
  private void initializeAccounts(int initialBalance) {
    while (numCustomers) {
      ...;
      Account a = new Account(i, n, s, c);
      accounts.add(a);
      numCustomers--;
    }
  }
  public void openAccount(Account a){
    accounts.add(a);
  }
  public boolean withdraw(int customerID, int amount) {
    if (...) {
      this.getCustomercustomerID).setBalance(
        this.getCustomer(customerID).getBalance() - amount);
      return true;
    } else return false;
  }
  public static void main(String[] args) {
    ...;
    Bank merchants = new Bank("Merchants", 100, 10000);
    Account a4 = new Account(1, "ABC Market", 1000000, 100000, 20000000);
    Account a5 = new Account(2, "CDE Outlet", 5000000, 300000, 150000000);
    merchants.openAccount(a4);
    merchants.openAccount(a5);
    ...;
    Account a = merchants.getCustomer(2);
    merchants.withdraw(a.getId(), 900);
  }
}
```

Figure 2. An example of a Java program.

The `Bank` class initializes a number of `Account` structures for its clients. On an `openAccount` event an `Account` reference is passed to the `Bank` object and it is added to the existing accounts list. The `Bank.withdraw(...)` method reduces the balance by the amount withdrawn. The `main` method creates instances of a bank and various types of accounts that are opened and operated on. Our analysis is targeted toward finding these instances and their dependences.

We have implemented an improved version of Spiegel's algorithm [20] (for detailed contrast see [2]). We use rapid type analysis (RTA) to compute the call graph and the program types. Then, for each method in the graph we compute the *class relations* by looking at field access and method call statements. A *usage* relation between two classes occurs when one class calls methods or accesses fields of another class. *Export* or *import* relations occur when new types may propagate from one class to another through field accesses or method calls.

Figure 3 shows the class relation graph for our example. We use the aiSee[1] tool for the visualization of the graph in the Visualising Compiler Graphs (VCG) format. The types are annotated with the `ST_` or `DT_` prefix to indicate static or instance (dynamic) parts of a class. The *use* relations tell that some classes occur in the context of other classes and their occurrence is noted by looking at the method calls,

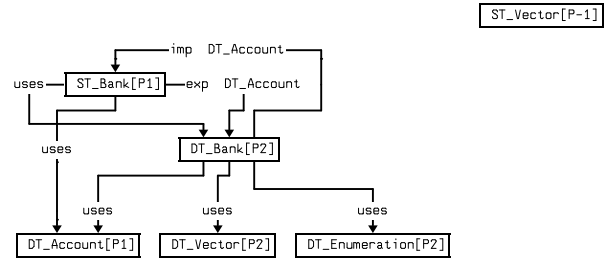1 A graph visualization tool from AbsInt. Available from http://www.absint.com/aisee.html.

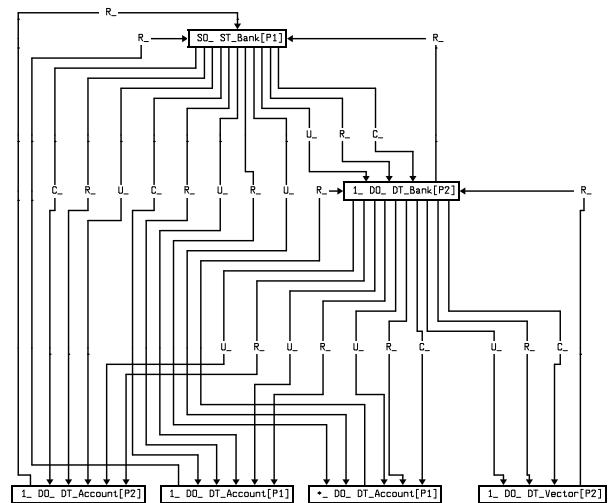**Figure 3. The class relation graph visualized with aiSee tool for vcg format.**

Figure 4. The object dependence graph visualized with aiSee tool for vcg format.

field accesses, and allocation statements. The *export* edge occurs due to the invocation of the `openAccount` method on the dynamic `Bank` class with an `Account` class as parameter. The *import* edge occurs due to the `getCustomer` invocation that returns a result of `Account` type.

Given the class relation graph, and the object set, we compute the relation between the corresponding objects (class instances). For each allocation statement, we add *reference* relations between the instance of the class where the allocation takes place and the newly created instance. We then create new references by matching the initial object references against the *export* and *import* relations between the corresponding classes. We iterate through all object triples and propagate references matching against the type relations until the algorithm reaches a fix point.

Figure 4 shows the object graph for our example. The

edges are labeled by *create*, *use*, *reference*. The objects are prefixed by `1_` indicating single instances (a `*_` prefix indicates summary instances of zero or more — i.e., created inside a control structure). The *reference* relation is redundant and only used for intermediate processing. We can safely abandon it. The *create* relation means that an object creates another object. The creation relation between object pairs is propagated to discover new *usage* relations from the class relation graph. Therefore, after the propagation, only the *usage* relation should matter for the partitioning: if an object `a` on abstract processor `Pa` *uses* an object `b` on abstract processor `Pb`, then communication may be generated. We also show the partition number within square parentheses for a two way partitioning transformation. For details on the actual algorithm and implementation, please refer to our technical report [2].

3. Graph Partitioning

The next transformation our system performs is the graph partitioning. As a result, this phase assigns a virtual processor number to each object.

A multi-constraint graph partitioning gives an optimal partitioning of the object dependence graph such as to minimize the cut, and thus communication, and to account for the resource constraints of each partition.

Finding an optimal multi-way partition for large graphs is an NP-complete problem (thus, no algorithm that solves the problem in polynomial time exists). However, many heuristic-based approaches exist [3, 12]. To our knowledge the most advanced multilevel partitioning scheme is Hendrickson et al.'s [7].

We use Metis' multi-objective, multi-constraint graph partitioning algorithms to partition the dependence graph. We model the resources for the object dependence graph as follows. Each object in the graph encapsulates data and computation. The amount of data it encapsulates characterizes the memory usage, while the amount of computations characterizes the CPU usage. The weight of a node is a vector that contains memory, CPU, and battery usage for the creation and usage of an object. An edge between two objects indicates a potential communication, if the objects were to reside in two different address spaces. The data that needs to be transferred between address spaces is the dependence data (i.e., field, method arguments or result). The weight of an edge is the amount of data that needs to be transferred due to a dependence.

We use static approximations of resource consumption to guide the static partitioning. The static approximations can be imprecise under the assumption that all objects have equal weights. In the future we plan to use simple heuristics; for example, objects created inside the loops can be considered "heavier" than single instance objects, etc.

```
Java:
public class Example
{
    int ex ( int b ){
        b = 4; // 1
        if (b > 2){ // 2
            b++; // 3
        }
        return b; // 4
    }
}
Quad:
BB0 (ENTRY) (in: <none>, out: BB2)
BB2 (in: BB0 (ENTRY), out: BB3, BB4)
1   MOVE_I              R1 int, IConst: 4
2   IFCMP_I             IConst: 4, IConst: 2, LE, BB4
BB3 (in: BB2, out: BB4)
3   ADD_I               R1 int, IConst: 4, IConst: 1
BB4 (in: BB2, BB3, out: BB1 (EXIT))
4   RETURN_I            R1 int
BB1 (EXIT) (in: BB4, out: <none>)
```

Figure 5. Turning a Java class into quads.

In our current implementation we have written a Java wrapper [10] for the Metis graph partitioning tool [14]. The wrapper implementation (including visualization capabilities) is about 10000 lines of code.

4. Code and Communication Generation

Once each object has been assigned to a virtual processor, the program can be distributed by mapping virtual processors to actual processing units at runtime. There are two issues related to the distributed execution. First, native execution in heterogeneous environments requires retargetable code generation. Second, correct execution requires communication to satisfy the remote dependences.

To address retargetable code generation we use the quad high-level intermediate representation to generate Abstract Syntax Trees (AST) and then use bottom-up rewrite system (BURS) [18] to emit code for a range of architectures (currently x86 and StrongARM).

To address communication generation, we use the dependence and partitioning information to classify objects as *local* and *dependent*. Local objects have no dependences on objects in different address spaces. Thus, they are treated as normal objects and no communication is generated for those. Dependent objects have dependences across address spaces and thus, messages are inserted to resolve these dependences.

4.1. Retargetable Code Generation

The input for this phase is the quad intermediate representation. The result is a generated set of compilers for various target machines. An example of the quad format is listed as Figure 5, along with the Java class that was used to generate the code.
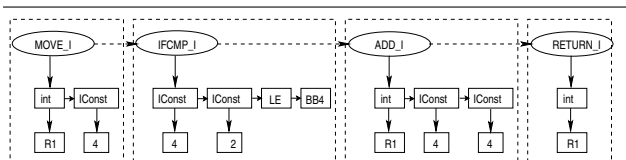
Figure 6. A Tree representation of the quads.

Abstract Syntax Tree. Once the quad source is established, the program is then turned into an Abstract Syntax Tree to act as the code generator front-end. The AST is structured such that each instruction acts as a root node, with instruction parameters represented as child leaves. The tree generator used is called ANTLR [16], and is a grammar parser similar to Yacc. A visual representation of this tree can be seen as Figure 6.

Because of the inherent simplicity in the quad format, it is feasible that a simple, linear parser be written from scratch and a code generator built on top of it. Though that approach may perform faster and can be more specialized to this task, using the tree allows extensibility. This would allow the code generator to be used with any intermediate representation or source language as creating a tree allows us to completely abstract the source.

Bottom-Up Rewrite Generation. After obtaining the tree representation of the source, the remaining work is done through the back-end and is handled through a method called Bottom-Up Rewrite Machine Generation, or BURM. This does two passes of the incoming AST: an initial pass to find a minimum-cost traversal, followed by a second pass that emits code based on the instructions represented in each node. The specific machine generator is called JBurg, a Java-based BURG (Bottom-Up Rewrite Generator) [5] that differs from other BURM implementations in that it traverses the tree employing dynamic programming pattern matching to satisfy goals. Two examples of machine code emitted by the BURG are as Figure 7.

4.2. Communication Generation

To generate communication, we generate partitions offline for 1, 2, ... nodes. This is a form of off-line rather than runtime specialization.

Each node in the object graph has a unique identifier that contains a virtual processor number. Communication is inserted only for dependent objects. That is, for each dependence relation *to* a remote object two calls are generated: a **send** call that packs the access type and associated data, and a **receive** call that fetches the response. For each dependence relation *from* a remote object, two calls are generated: a **receive** call that processes the access type and associated data and a **send** call that sends the results of the access back.

```
x86:
    mov eax, 4 ; 1
    cmp 4, 2 ; 2a
    jle BB4 ; 2b
    mov eax, 4 ; 3a
    add eax, 4 ; 3b
BB4:
    ret eax ; 4
StrongARM:
    mov R1, #4 ; 1
    cmp #4, #2 ; 2a
    ble BB4 ; 2b
    add R1, 4, 4 ; 3
.BB4
    mov PC, R14 ; 4
```

Figure 7. Machine code for two separate architectures

```
Original byte-code:
    13: aload //load Account object
    14: invokevirtual Account.getSavings:()
Transformed byte-code:
    13: aload //load DependentObject object
    14: ldc INVOKE_METHOD_HASRETURN (int)  //access type
    16: ldc "getSavings" //load method name
    18: aconst_null //no method argument for getSavings()
    19: invokevirtual DependentObject.access
    22: checkcast Integer //cast to return type
    25: invokevirtual Integer.intValue //get primitive value
```

Figure 8. The transformation for method invocation `account.getSavings();`**.**

The dependences handled by our current implementation are object accesses, including field accesses, and method invocations. For each dependent object that is referred from remote, there is a corresponding `DependentObject` that performs Message Passing Interface (MPI) communication with the home node of the referring object. Distributed dependences are therefore transformed to accesses to `DependentObject` instances.

Figure 8 illustrates the original and transformed bytecode snippets for method invocation `account.getSavings()`. The transformation for method invocations performs three tasks: prepare the arguments for the `DependentObject` access, prepare the arguments (in a `LinkedList`) for the original method call, and cast the return value (`Object` type) to the appropriate class type or primitive value. The transformation for field accesses are similar.

The remote instantiation of a dependent class is translated to an instantiation of a `DependentObject`, which in turn will communicate via MPI messages to the home node of the dependent object. The home node will then create the object. Figure 9 demonstrates the transformation of `new Account(i, n, s, c)`. The information passed to the MPI message for distributed instantiation and com-

```
Original byte-code:
   35: new Account
   38: dup
   39: iload_2     //i
   40: aload_3     //n
   41: iload 4     //s
   43: iload 5     //c
   45: invokespecial Account."<init>"
Transformed bytecode:
   35: new DependentObject
   38: dup
   39: iload_2     //i
   40: aload_3     //n
   41: iload 4     //s
   43: iload 5     //c
   //....
   // prepare the constructor arguments
   //....
   105: ldc 0 (int) //location of Account, Node0
   107: ldc "Account" (String)
   109: aload 6       //constructor arguments in a list
   111: invokespecial DependentObject."<init>"
```

**Figure 9. The transformation for `new Account(i, n, s, c);`**

prises of the class name and the arguments to the class constructor.

The quality of communication generation is directly influenced by the quality of dependence analysis. Our analysis is type-based and thus, not very precise. More precise dependence information makes use of points-to information [19] in the context of speculative multithreading. In addition, there are several communication optimization techniques that can be applied to optimize communication generation: message aggregation, hoisting communication out of the loop, asynchronous communication, overlapping communication and computation, data replication, and early prefetch. Many of these techniques cannot be used with request/response communication style like RPC or RMI. In contrast, we use message exchange communication to reveal more optimization opportunities.

5. Distributed Execution

The distributed target code partitions are executed within the MPI enhanced runtime environment. Currently we use JVM hosted execution rather than native execution. Even though the retargetable code generation component is fully implemented, it was easier to use normal JVM since our current experiments are conducted on resource-rich x86 platforms. Also, the use of JVM does not affect our current distributed execution evaluation (speed-up measurements).

In our current implementation, on each node there are three supporting services: the `MPI` service, the `ExecutionStarter` service, and the `Message Exchange` service. Figure 10 depicts this organization of the runtime services for distributed execution. The `MPI` service sets up the necessary MPI working environ-
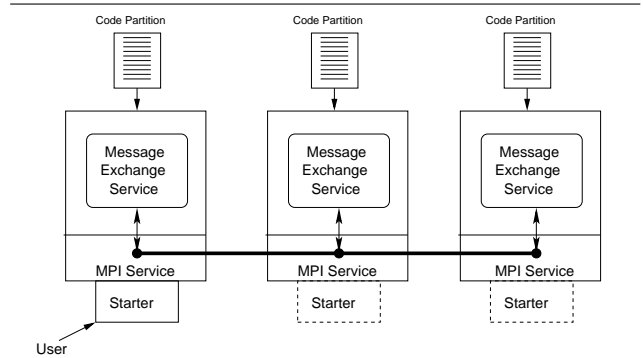
Figure 10. The organization of runtime services for distributed execution.

ment — such as groups, communicators, and the communication context.

The `Execution Starter` service starts the application by invoking the `main()` method of the application class. Only one copy of `Execution Starter` needs to be active on the processor node in the distributed execution environment where the user initiates the application.

The core of this MPI-aware runtime support is the `Message Exchange` service. This service processes all the `send` and `receive` MPI communication generated from the object dependence information. The `Message Exchange` service uses two supporting data structures. One is the `DependentObject` and the other is the exchanged `Message`. The runtime uses the `DependentObject` (implemented by a Java class) to indicate an object that has dependence relations to another partition.

Each dependent object contains the following information: its class type, the identifier of the partition (node) that hosts the object, and its unique identifier in that partition (node). A message (packed in a `Message` structure) exchanged between two dependent objects across two nodes contains the object identifier of the receiver of the communication and the relevant dependence data. The `Message Exchange` service passes objects between nodes using a streamed format.

We currently identify two types of messages: `NEW` and `DEPENDENCE` for object instantiation and data dependence. We are in the process of defining more precise dependence relations (e.g., *read after write*), and discriminating further between messages.

6. Profiler

We have built a profiler that collects statistics indicating the resource consumption of a program during runtime.

The profiler is built on top of the Joeq compiler and virtual machine. The profiler works either through instrumentation or sampling. Some of the metrics can be implemented using either technique. In these cases, the instrumentation is useful as a baseline for comparison of the accuracy of the sampling. There are four basic categories of runtime application behavior we are interested in: CPU, memory, battery, and communication (i.e., network) usage. To measure these four basic categories, we have currently implemented six metrics: method duration, method frequency, hot methods, hot paths, memory allocation, and dynamic call graph.

The **method duration** metric measures the amount of time each method took to execute. The metric was originally implemented by overloading the method invocation process of the built-in native [2] interpreter. The time of entry and exit of each method (both system-level and user-level) are recorded in a profiling class. Unfortunately, due to problems within Joeq itself, this metric on our test benchmarks had to be measured with Java source level instrumentation. See Section 7.3 for details.

The **method frequency** metric measures how often each method is invoked. This metric can also be used as a less expensive substitute for the method duration metric. A counter is associated with each method that kept track of the number of invocations. However, also like the method duration metric, source level instrumentation had to be performed instead.

The **hot methods** metric minimizes the overhead of the previous metric by using sampling. For each native thread Joeq spawns it also attaches a separate native interrupter thread. The interrupter thread's main task is to signal the thread queue when to switch threads. This provides a convenient approach to sampling; simply pass control from the interrupter thread to the profiler at each scheduling time quantum. The profiler then obtains the currently executing method by reading the call stack of the thread and recording the top stack frame.

The **hot paths** metric goes a level above the hot methods metric in its scope and measures the hottest execution paths through the application. We extend the hot method technique, and we sample the entire call stack instead of sampling only the top stack frame.

The **memory allocation** metric is implemented by directly modifying the internal Java virtual machine system code of Joeq. By overloading some of the methods that implement memory allocation, we can estimate the memory profile of the application without performing instrumentation. Unfortunately, this metric is currently only a very rough approximation, but we are confident that much better accuracy will be achieved in the near future.

---

2   "Native" in context of Joeq means it bootstraps itself into a fully functional JVM without the need for a host JVM to support it.

benchmark	size			CRG			ODG		
	#C	#M	KB	#N	#E	EC	#N	#E	EC
create*	14	28	13	17	6	2	210	632	82
method*	6	35	10	12	10	2	9	32	2
crypt*	6	45	12	13	13	3	11	33	1
heapsort*	6	42	10	13	13	3	11	33	2
moldyn*	8	48	17	12	15	2	9	32	2
search*	9	57	17	14	23	3	6	20	3
cmprss**	38	295	160	36	42	1	32	107	2
db**	32	299	155	32	26	2	49	164	8

* Java Grande benchmarks: JGFCreateBench and JGFMethodBench (section 1), JGFCryptBench and JGFHeapSortBench (section 2), JGFMolDynBench and JGFSearchBench (section 3).

** SPEC JVM98 benchmarks: _201_compress, and _209_db.

Table 1. The size of the benchmarks (number of classes, methods, and KB) and the sizes of the resulting graphs (the number of nodes, edges, and the edgecut for both CRG and ODG).

The **dynamic call graph** metric shows the methods that actually got called in a specific application instance. It was measured using sampling. It also makes use of similar data as the hot paths metric, but processes the data in a different manner to actually construct the dynamic call graph.

7. Evaluation

We have implemented a functional infrastructure prototype that realizes the components presented in the above sections. We evaluate the functionality and the performance of our prototype with a set of benchmarks from Java Grande benchmark suite and SPEC JVM98 (see Table 1). In our experiments the networked configuration includes a service node, 1.7GHz Pentium III machine (512MB RAM, SuSE9.1), and another computation node, a 800MHz Pentium III (384MB RAM, Redhat 9.0). Both nodes run JDK 1.4. The two nodes are connected via 100M Ethernet. At the time of this publication we did not have access to other networked configurations and we only experimented with the few computers we had access to. However, in the future, we plan to set up a network consisting of multiple nodes with significant differences in resources and configurations.

7.1. Dependence Graph Construction

Table 1 shows the sizes of the original benchmarks as well as the resulting class relation graph (CRG) and object dependence graph (ODG) for each benchmark. The *edgecut* is the number of edges that straddle partitions. Currently we use the class relation graph partitioning to distribute the program.

benchmark	construct		partition		rewrite
	CRG	ODG	TRG	ODG	
create	2043	3056	7	12	271
method	1704	53	7	6	202
crypt	1715	40	7	7	209
heapsort	1615	54	6	7	193
moldyn	1903	114	6	6	215
search	1868	49	7	7	204
compress	2305	100	6	7	285
db	2434	99	10	7	280

Table 2. The execution time breakdown in code distribution. The columns indicate the construction time, the partitioning time, and the bytecode rewriting time

Figure 11. Performance comparison of centralized and distributed executions.

The execution times for graph construction and distribution transformation are shown in Table 2, in milliseconds. We can see that the static analysis of the class relations is in the order of seconds. This is because the process to extract high-level dependence information from the low-level bytecode format is computation and time consuming. However since this process only happens once at compiler-time, it is not as crucial as the other phases in the dynamic repartitioning process — ODG construction, partitioning, and code rewriting. In these latter phases, only partitioning has to be completely re-executed in each adaptive iteration. ODG construction and code rewriting can be both adjusted incrementally. Since the partition time is only about 10ms, we believe that the results are promising for our future plans of incorporating adaptive repartitioning. Also, *Create* benchmark has an unusual long ODG construction time. This is because it creates a large amount of objects which substantially complicate the object graph.

7.2. Distributed Execution

To evaluate the performance of the distributed execution runtime, we compare the distributed execution time of the transformed benchmarks with the execution time of the original sequential benchmarks on the 800MHz Pentium III machine. The execution speedup is depicted in Figure 11. The distributed execution shows comparable or improved performance (79.2% to 175.2%) with the original sequential execution. The results are promising, since without any further optimization the distributed execution results in very little overhead (in *Method* and *Compress*), or speed-up. Since we currently use a suboptimal naive partitioning, it is expected that further performance gain will be achieved if optimization is introduced to the distribution infrastructure in our future work.
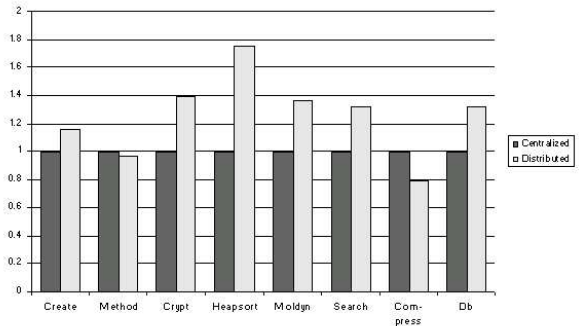
7.3. Profiling

We evaluate the profiler for a a subset of the Java Grande Forums benchmarks. For the baseline measurements, Joeq runs each of the benchmarks with all the profiling code compiled in, but not enabled. Then each of the profilers is enabled in turn. The tests were conducted on an AMD Athlon XP 2000+ (1.67 GHz) with 512 MB RAM running Windows XP. In each of the tests, Joeq was allocated a maximum heap-size of 1024 MB.

Table 3 shows the total execution times for each of the benchmarks and profilers. The average overhead for all the profilers is 21.94%. A general trend is that metrics which were measured with instrumentation overall incurred notably higher overhead than did the others, which used either sampling or modification of the JVM system code. The hot paths, dynamic call graph, and memory usage metrics all incurred about equal levels of overhead, approximately 14-20%. The most impressive results came from the hot methods metric, which at approximately 4% is a very good result.

8. Related Work

There are two types of automatic distribution compilers or virtual machines available: automatic distribution to exploit data parallelism in scientific programs and automatic partitioning of Java programs to relieve resources on constrained devices.

Automatic Distribution of Data Parallel Programs. Automatic parallelization is one research area that has investigated the partitioning problem mainly for scientific programs typically targeting a significant reduction in CPU or memory consumption [8, 13, 1, 6, 11]. There are two main differences between partitioning for scientific applications and our work. First, most of the previous work focuses on array partitioning, or loop iteration partitioning for

Test/Metric	Baseline	Hot Paths	Dynamic Call Graph	Hot Meth- ods	Method Du- ra- tion	Method Fre- quency	Memory Us- age
CreateBench (int [])	4.406	5.125	5.375	5.468	4.734	5.937	9.718
CreateBench (long[])	18.250	28.046	28.640	19.281	25.140	31.062	35.000
CreateBench (float[])	4.468	6.437	5.906	4.265	5.015	4.659	6.015
CreateBench (Object[])	2.156	2.421	2.468	2.328	2.296	2.203	2.281
CreateBench (Custom[])	10.718	12.687	12.500	11.484	11.875	11.234	51.406
MethodBench	196.187	212.140	222.359	202.281	323.437	248.156	198.937
FFTA	32.187	37.609	40.765	33.812	35.781	36.546	34.312
HeapSortA	3.906	4.296	4.968	4.281	17.297	14.328	3.968
MolDynA	48.234	53.062	57.390	50.234	51.375	51.750	50.125
MonteCarloA	48.734	59.859	58.890	51.015	75.194	60.234	49.671
Total:	369.734	421.682	439.261	384.449	552.144	466.109	441.433
Overhead:	0.00%	14.05%	18.80%	3.98 %	49.34%	26.07%	19.39%

Table 3. The profiler evaluation. Each row is the individual benchmark, while each column is the name of the profiler enabled. The last row is the total time it took to execute all the benchmarks. The times are given in seconds. The baseline column is the execution times with all the profiling code compiled in but not enabled.

scientific programs. We address general program distribution, where all the objects in a program are of interest. Second, the main objective for partitioning in scientific programs is to speedup execution, either on distributed or on shared memory machines. Our design choices are motivated by the ability to model multiple resources and study their interaction. Then, the general distribution can be specialized at runtime depending on resource priorities and actual environment.

Automatic Distribution of Java Programs. Java-Party [17] extends Java with *remote objects*. The objective is to provide location transparency in a distributed memory environment. In contrast, we achieve the transparency effect without extending Java syntax. However, we do not give the user any control over distribution.

Messer et al.'s approach, though entirely dynamic, has an objective that more closely matches our own [15]. The goal is to transparently off-load services to relieve memory and processing constraints on resource-constrained devices. The main difference is the handling of object references. In this approach each JVM maps all other JVMs references, and thus it results in a *replicate all* strategy. Our approach is partly static, and it considers just some of the interactions between objects (cross processor).

Another approach, similar to the distributed shared memory paradigm, is to implement a distributed JVM as global object space [4]. We achieve the same transparency effect at hopefully lower cost, since we distinguish between local and remote accesses.

J-orchestra [21] transforms Java bytecode into distributed Java applications. This is also an abstract shared memory implementation. The communication is synchronous only — i.e., RMI. To exploit asynchronous communication, we use automatically generated point-to-point messages.

Pangaea [20] is a system that can distribute Java programs using arbitrary middleware (Java RMI, CORBA) to invoke objects remotely. The system is based on the original algorithm by Spiegel which was the basis for our own extended algorithm [2]. Pangaea's input is a centralized Java source-code program. The result is a distributed program underlying the synchronous remote method invocation communication paradigm. Our approach starts from Java bytecode and targets a flexible distribution model (i.e., allows the exploitation of concurrency and asynchronous communication) in a program.

Coign [9] is also a system that strives to automatically partition binary programs (built from COM components) for optimal execution. Coign is designed to handle 2-way partitioning only (between two nodes) for client-server distributions. Also, the distribution is fully dynamic, based on profiling history. We combine static analysis with off-line distributions in a general, multi-way partitioning.

9. Conclusion

This paper presented the design and implementation of a research compiler and runtime infrastructure for automatic program distribution. While not all programs can benefit from automatic distribution, we believe that it is important to be able to model the resources of a program and study the effect of distribution on program behavior with respect

to resource consumption. The motivating factor to our design was flexibility and modularity. Thus, we expect each of the techniques we presented to evolve as more experiments are conducted.

Our design is based on two key ideas: find the dependences between the objects in a program and use this information to automatically generate communication. We have shown how we cast the resource modeling and program distribution problem into an optimal graph partitioning problem. We model the resources as weights on the dependence graph and then experiment with multiple resource priorities and constraints. We have presented the code generation phase as two separate parts: platform independent code generation and communication generation.

We have also described a profiler system that allows us to collect information about the program behavior and eventually, be able to redistribute the program according to the actual access patterns and resource requirements. Our present infrastructure only handles static partitioning. While dynamic repartitioning is the goal of our next design iteration, it does not influence the design of the infrastructure presented in this paper.

Finally, we have presented results on each of the techniques that we have introduced. The results indicate that partitioning takes little time and the computed dependence graphs are within manageable sizes. We have also shown that without any further tuning, the distributed execution results in either a very small overhead or a speed-up. Finally, we have evaluated our profiler system in terms of the incurred overhead as well as collected data.

References

[1] C. Ancourt and F. Irigoin. Automatic Code Distribution. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.

[2] R. E. Diaconescu, L. Wang, and M. Franz. Automatic distribution of java byte-code based on dependence analysis. Technical Report Technical Report No. 03-18, School of Information and Computer Science, University of California, Irvine, October 2003.

[3] S. Dutt. New faster kernighan-lin-type graph-partitioning algorithms. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 370–377. IEEE Computer Society Press, 1993.

[4] W. Fang, C.-L. Wang, and F. Lau. Efficient global object space support for distributed jvm on cluster. In *International Conference on Parallel Processing*, Vancouver, Canada, August 2002.

[5] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, 1992.

[6] M. Gupta and P. Banerjee. Paradigm: a compiler for automatic data distribution on multicomputers. In *Proceedings of the 7th international conference on Supercomputing*, pages 87–96. ACM Press, 1993.

[7] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 28. ACM Press, 1995.

[8] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, 1992.

[9] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *Operating Systems Design and Implementation*, pages 187–200, 1999.

[10] Java interface to metis graph partitioning and visualization tool. Available at: http://www.cacr.caltech.edu/ roxana/code/jmetis.tar.gz.

[11] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proceedings of the 1995 conference on Supercomputing (CD-ROM)*, page 76. ACM Press, 1995.

[12] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, February 1970.

[13] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[14] Metis family of multilevel partitioning algorithms. Available at: http://www-users.cs.umn.edu/ karypis/metis/.

[15] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, T. Giuli, and X. Gu. Towards a distributed platform for resource-constrained devices. In *Proceedings of 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*. IEEE, July 2002.

[16] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.

[17] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[18] T. A. Proebsting. Burs automata generation. *ACM Trans. Program. Lang. Syst.*, 17(3):461–486, 1995.

[19] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 77–90. ACM Press, 1999.

[20] A. Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, Fachbereich Mathematik u. Informatik, Freie Universitat Berlin, 2002.

[21] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP 2002 - Object-Oriented Programming: 16th European Conference Malaga, Spain*, volume 2374/2002 of *Lecture Notes in Computer Science*, pages 178–204. Springer-Verlag, June 2002.

[22] J. Whaley. Joeq: A virtual machine and compiler infrastructure. In Proceedings of the Workshop on Interpreters, Virtual Machines, and Emulators, San Diego, CA. Pages 58-66., 2003.